Project 5

Computer Science 2334
Fall 2008

User Request:

"Create a GUI Word Guessing Game with Frequency Displays that Handles I/O Exceptions."

Milestones:

1. Create a graphical user interface to allow the user to input information into the game and to present the user with information about the state of the game. *15 points*
2. Create a dictionary to store words and scores, as done in Project 4, but remove from the dictionary information on languages (in this project, words will be considered regardless of language) and on frequencies (in this project, frequencies will be outside the dictionary). *5 points*
3. Create a recursive function to create all of the permutations of an array of letters and save them in an appropriate data structure. *20 points*
4. Use the MVC paradigm as exemplified by the **ExampleCleanMVC** code from lab 8 and as discussed in class. *20 points*
5. Handle I/O exceptions for reading in the dictionary from a text file. *10 points*

- ► Develop and use a proper design. *15 points*
- ► Use proper documentation and formatting. *15 points*

Overview:

An important skill in software design is extending the work you have done in a previous project. For this project you will rework Project 4, modifying the graphical user interface and the code to allow the user to play a simple game. For this program you may reuse some of the classes that you developed for your previous projects, although you are not required to do so.

For this project, as with your previous projects, you will put together several techniques and concepts learned in CS 1323 and some new techniques to make a new application. This application will be a simple word search game where the user looks at a collection of letters and tries to recognize words within it. Note that much of the code you write for this program could be reused in more complex applications, such as a spell-checker or an actual dictionary (with definitions).

The Game:

The object of the game is for the user to guess all the words that can be made from the letters in a small random collection.

<u>Brief Instructions:</u>

1. The user will enter the number of letters (a-z) the game should provide to him or her. Call this number *n*.
2. The game will generate a random collection of *n* letters. Call this collection the "starting letters."

3. The game will search its dictionary for all words it contains of length *n* or shorter that can be made from the starting letters. Call this set of words the "matching words."
4. The game will display to the user a histogram of black rectangles showing frequencies of the lengths of the matching words.
5. The game will display to the user the starting letters.
6. The user will enter a sequences of letter into the game's GUI and hit return. This is known as "guessing."
7. The user will continue guessing until he or she correctly guesses all matching words or gives up.

Complete Instructions:

1. The user will enter the number of letters (a-z) the game should provide to him or her. This number must be equal to or greater than four. Call this number *n*.
2. The game will generate a random collection of *n* letters. Call this collection the "starting letters." The same letter may appear more than once in the starting letters and is counted once for each time it appears. For example, the collection "a b h e l a r" is a possible collection of starting letters for $n = 7$.
3. The game will search its dictionary for all words it contains of length *n* or shorter that can be made from the starting letters. Call this set of words the "matching words." For each word, each letter from the starting letters can be used as many times as it appears among the starting letters or fewer. For example, from the starting letters in step 2 above, you could use up to one e and two a's in a single matching word. Letters may be re-used in subsequent matching words. For example, both "bar" and "bear" could be made from the starting letters above, even though there is only one b and one r among the starting letters. If there are fewer than five matching words that can be generated from the starting letters, the game will return to step 2 and generate a new collection of starting letters of size *n*.
4. The game will display to the user a histogram of black rectangles showing frequencies of the lengths of the matching words. In your previous assignments, your histograms could have buckets containing more than one value. For example, you could choose to have a single bucket that contained the number of occurrences of words of length 1 through 10. In Project 5, however, your histograms may not group more than one length value into a single bucket.
5. The game will display to the user the starting letters.
6. The user will enter a sequences of letter into the game's GUI and hit return. This is known as "guessing." When the user guesses, one of two results will occur:
    1. If the sequence of letters guessed by the user is a matching word that has not been previously entered by the user, the program will make a pleasant dinging or binging noise and flash a green graphic at the user. It will also add the guessed word to a list of correctly guessed words that is visible to the user (and which should initially be empty) and display the user's current score, which is calculated by adding up the scores of all words guessed so far. In addition, it will draw a histogram of green rectangles showing the frequency of the lengths of guessed words "in front of" the histogram of matching words. (Here "in front of" simply means that the green rectangles will replace the black ones as the user correctly guesses more matching words.)
    2. If the sequence of letters guessed by the user is not a matching word, or is a matching word

that has previously been entered by the user, the program will make an annoying beeping or buzzing sound and flash a red graphic at the user.

7. The user will continue guessing until he or she correctly guesses all matching words or gives up. If all matching words are correctly guessed, your program will play a happy tune and tell the user that he or she has won. If the user gives up (by pressing a button labeled "Give Up") your program will add all of the unguessed matching words, in red, to the list of guessed words displayed to the user. In either case, your program will then present the user with a dialog asking if the user wishes to play again.

The Code:

Model, View, Controller

You should organize your code using the same Model, View, Controller paradigm you were required to use for Project 4. However, in this case you must determine for yourselves what classes this will entail and you should name them and organize them appropriately.

GUI:

Your graphical user interface should support the game described above. Exactly what the GUI will look like and the classes you use to create it are up to you, as long as they are logical, easy to understand, and follow established conventions. In your write up, you should include a diagram explaining what the GUI looks like and how it works (as was provided for you in Project 4).

Importing:

Your software will read in a text file for the dictionary of words. This text file will contain only words, no scores or languages. As it reads in each word, it will calculate a score for it as follows. First, it will sum up the score of the individual letters.

The following letters are worth one point each: e, t, a, o, i, n, s, h, r, d, l, and u.
The following letters are worth two points each: b, c, f, g, k, m, and p.
The following letters are worth three points each: j, v and w.
The following letters are worth four points each: q and y.
The following letters are worth five points each: x and z.

Second, it will multiply the sum score by the length of the word. It will store this in the word's score field.

I/O Exception Handling:

Your program should include `try-catch` blocks as appropriate to handle possible I/O exceptions. In particular, `main` should not re-throw any I/O exceptions; they should all be dealt with inside your code.

Recursion:

Your program will need to find all words in your dictionary that can be made from the starting letters. To do this, your program will use recursion to generate all possible permutations of the starting letters and check for each one in the dictionary. (Note that permutation has several similar but slightly different meanings. You should determine the correct meaning here from the description of the game given above.) When it finds a permutation in the dictionary, it will add it to the collection of matching words.

Note that your recursive permutation method will run out of stack space if asked to find permutations of large collections of starting letters. Find the smallest *n* for which your program runs out of stack space while doing permutations and report that value in your write up. Does it appear to be consistent, regardless of the starting letters generated? *Explain* your answer.

Implementation Issues:

This project gives you more flexibility in design than your previous projects. If you have questions about your design, please work through them with your teaching assistants and instructor before moving on to implementation.

Due Dates and Notes:

Your revised design and detailed Javadoc documentation are due on **Monday, November 24th**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your revised UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation, and a hardcopy of the stubbed source code **by 5:00pm on Tuesday, November 25th**.

The final version of the project is due on **Monday, December 1st**. Submit the project archive following the steps given in the submission instructions **by 9:00pm**. Submit your final UML design on *engineering paper* or a hardcopy using UML layout software, a hardcopy of the index page of your Javadoc documentation and a hardcopy of the source code at the **by 5:00pm on Tuesday, December 2nd**.

You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you must give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

As noted in the syllabus, you are required to work on this programming assignment in a group of at least two people. It is your responsibility to find other group members and work with them. The group should turn in only one (1) hard copy and one (1) electronic copy of the assignment. Both the electronic and hard copies should contain the names and student ID numbers of **all** group members. If your group composition changes during the course of working on this assignment (for example, a group of five splits into a group of two and a separate group of three), this must be clearly indicated in your write-up, including the names and student ID numbers of everyone involved and details of when the change occurred and who accomplished what before and after the change.

Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project. For example, if you have three functions in your program and one function was written by group member one, the second was written by group member two, and the third was written jointly and equally by group members three and four, both your write-up and the comments in your code must clearly indicate this division of labor. Giving improper credit to group members is academic misconduct and grounds for penalties in accordance with school policies.